> 小小的感悟：很幸运能够在前端工程化的浪潮之中加入了前端的开发的事业，但是在学习的过程中，能明显感觉到，我的知识很难形成一条连贯的学习链条，很多知识总有一种“无源之水”的感觉。今日有幸能够接触到前辈们为前端工程化作出的各种努力，🙏感谢前辈们的付出，给了我这种“小码农”参与前端建设机会。

## 早期的前端

我猜早期的前端技术标准根本没有预料到前端行业会有今天的这个规模，设计上才会出现那么多的缺陷。比如，早期的web就是单一的浏览器请求获取资源展示页面，js也就局限在做一些表单验证等。（可以见的，对未来的预期有时候能决定发展上限，程序员对未来的预期不应该仅仅局限在娶个老婆生个娃什么的，too simple）

## 转折点

一个重要的转折点在于浏览器的网络进程对ajax支持的出现，他使得js脚本可以异步的发送请求，根据响应的数据来更新局部的DOM对象，而非整个页面，这使得页面有发展成为一个可拓展的应用的可能性。（依然记得小时候打开网页可以玩4399的兴奋感）

## 转折点之后

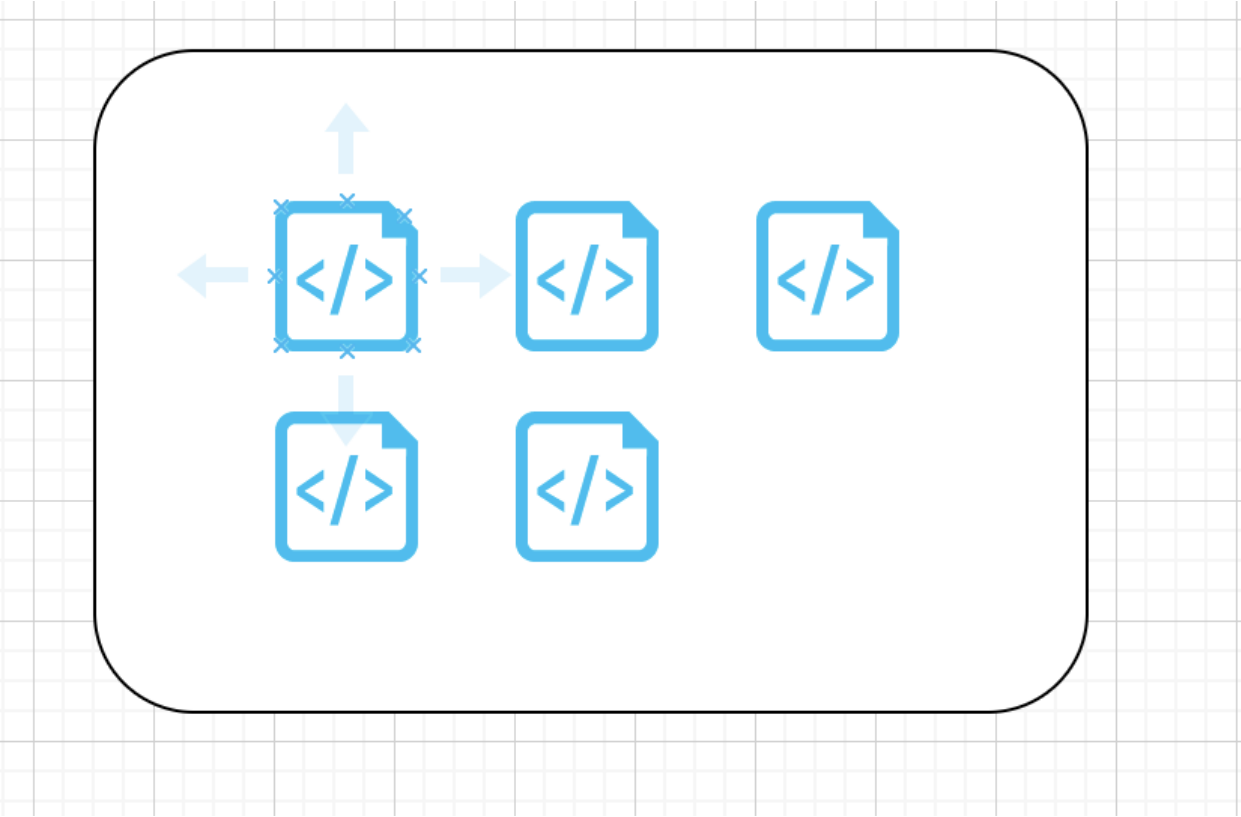
紧接着，代码体积就发生的巨大的膨胀，对项目的维护成本和要求也越来越高。这也催生了模块化开发的思想。后端的工业化早就催生了各种各样的设计模式，规范的开发流程。前端就显得有也仓促上马的意思，因为前端的语言、理论基础并不足以支撑起前端的工程化。比如，早期的js语言本身并不支持前端的工程化。

## 模块化的演进过程

阶段一：
文件各分方式，规定每个文件就是一个模块，它的缺点十分明显，会污染全局变量，容易命名冲突。并且模块的加载顺序必须按照一个特定的顺序，人为操作容易失误，不够人性化，不像是程序员干的事，容易被笑话。
缺点：

- 全局的作用域污染
- 没有私有空间
- 容易命名冲突
- 无法管理依赖关系

- 难以维护



阶段二：

每个模块向外暴露一个对象

```
        windows.moduleA = {}
        windows.moduleB = {}
        //...
```

虽然每个模块只允许暴露一个对象，但是window对象上还会造成污染，也会有命名冲突的风险。

阶段三：
立即执行函数，带了私有成员的概念，解决了全局作用的污染和命名冲突的问题，同时也可以利用参数的传递，可以解决模块之间依赖的关系

```
(
        function($) {
                var name = "module-a";
                function f1 () {} // 内部的私有空间
        }
)(JQuery) // 对JQuery的依赖
```

## 模块化还未解决的问题

1. 模块的加载问题，模块都是通过script标签人工写入的，这是不可控的，script标签的顺序，必须按照特定顺序，维护起来太麻烦。
2. 通过约定的方式来实现的模块化，很难统一，没有行业级别的标准，很难使得全球人民共同奋斗，标准都不一致，怎么协作呢？怎么共享轮子呢？

## 不成熟的方案

1. 按需加载
2. Common JS的出现：common js是在node中的规范，但是common js的加载方式同步加载，直接应用到浏览器端会阻塞浏览器渲染，引入新的性能问题。
3. AMD：专门为浏览器端定义的 异步模块定义规范，require.js

## 现在如今模块化

现在随着模块化的快速普及和发展，两种模块的标准已经明显胜出，并将继续发展下去。

1. 前端偏向使用 es module (ES6+)
2. node 环境中 偏向使用 common JS

# 新的挑战

1. ES6 的支持并不是100%，有的老旧浏览器并不能有效的支持ES6。
2. ECMA发展速度很快，但是浏览器的支持并没有提供很快的支持。
3. 工程的需求仅仅是在开发阶段，我们将大量的文件拆分开来，对于浏览器而言，浏览器需要发起多次http请求获取资源，效率不高。